1-1-2002

Automatic parallelization tools and their applications

Jaekyu Cho
Iowa State University

Automatic parallelization tools and their applications

by

Jaekyu Cho

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Suresh Kothari, Major Professor
Daniel Berleant
Simanta Mitra

Iowa State University

Ames, Iowa

2002

Graduate College

Iowa State University

This is to certify that the master's thesis of

Jaekyu Cho

Has met the thesis requirement of Iowa State University

Signatures have been redacted for privacy

TABLE OF CONTENTS

LIST OF FIGURES v

LIST OF TABLES vi

ACKNOWLEDGEMENTS vii

ABSTRACT viii

1. INTRODUCTION 1

2. RELATED WORKS 4

3. SOFTWARE ENVIRONMENT 7

4. PARALLELIZATION OF MM5 PROGRAMS 9

4.1 USE OF PARAGENT 9

4.2 PARALLELIZATION STPES 11

4.3 DEBUG 14

5. PERFORMANCE RESULTS 21

5.1 PNNL MM5 21

5.2 PERFORMANCE EXPERIMENTS WITH A FDM BENCHMARK 22

5.2.1 PROBLEM SIZE AND PERFORMANCE VARIATIONS 23

5.2.2 COMPARISON OF PC CLUSTERS AND OTHER PLATFORMS 26

6. LOAD BALANCE 28

7. COMMUNICATION LIBRARY 31

7.1 DATA DECOMPOSITION 32

7.2 LOOP INDEX TRANSFORMATION 34

7.3 DISTRIBUTION OF ARRAY 35

7.4 NEIGHBORHOOD COMMUNICATION 38

7.5 APPENDING ARRAY 41

8. CONCLUSIONS 44

REFERENCES 45

LIST OF FIGURES

Figure 1 Comparison of PC Clusters and Other Computing Platforms 27

Figure 2 Structure of an unbalanced array 28

Figure 3 Speedup before and after reducing load imbalance 30

Figure 4 Cyclic and Block Decomposition 32

Figure 5 Examples of Block Decomposition 33

Figure 6 Distributing an array for 1-D Parallelization 36

Figure 7 Distributing an array for 2-D Parallelization 37

Figure 8 Neighborhood communication for 1-D Parallelization 39

Figure 9 Neighborhood communication for 2-D Parallelization 40

Figure 10 Appending an array for 1-D Parallelization 42

Figure 11 Appending an array for 2-D Parallelization 43

LIST OF TABLES

Table 1 Execution Time (sec) of 2-D Parallel MM5 on 64-node PC Cluster 5

Table 2 Execution time, speedup, and efficiency for the PNNL MM5 22

Table 3 Execution time(sec), Speedup, Normalized and incremental speedup 25

Table 4 Analysis of load imbalance due to irregular distribution of elevation bands .. 30

ACKNOWLEDGEMENTS

I mostly thank my major professor, Dr. Suresh Kothari, for his guidance and assistance.

I am grateful to Dr. Daniel Berleant and Dr.Simanta Mitra for being my committee members.

I thank Dr. Don Heller for his help and providing Rabbit.

I appreciate the support of Pacific Northwest National Laboratory (PNNL).

ABSTRACT

In parallelizing huge legacy codes such as NCAR/Penn State MM5, a proper software environment is critical for reducing the time and effort. This thesis presents an empirical study of automatic parallelization based on the NCAR/Penn State MM5 model, the Pacific Northwest National Laboratory (PNNL) version of MM5 and a FDM benchmark program. ParAgent, a tool for automatic parallelization, Vis5-D a visualization tool, a web-based monitor, and Rabbit, a performance analysis tool were used in this study. In addition, a high-level communication library was developed to complement the use of ParAgent.

Performance is one of the most important aspects of parallelism. We tested different types of networks and PC clusters to see how the communication between processors affects performance. Also, we put some efforts in analyzing and reducing load imbalance.

1. INTRODUCTION

Climate modeling is highly computation and data intensive, and can clearly benefit from the application of parallel computing. For example, a simulation of 24 hours of climate using the NCAR/Penn State mesoscale meteorology model MM5 [1][6] requires about an hour on a powerful workstation. To study global warming, long-term simulations ranging over hundred years are necessary. Moreover, because of the lack of sufficient computing capability coarse spatial grids are used. Parallel computing can provide the capability necessary to perform long-term simulations and to improve accuracy by allowing the use of finer grids.

The Knowledge-Centric Software (KCS) Laboratory at Iowa State University is engaged in developing a comprehensive software environment to facilitate parallel computing. Included in the environment are tools for automatic parallelization, performance analysis, visualization and a web interface for monitoring a cluster.

Automatic parallelization is critical for legacy codes because manual parallelization is time consuming, tedious and prone to human errors. It is also one of the most difficult problems in parallel computing. Also, parallelization of a legacy code is not a one-time activity. Many legacy codes continually evolve as scientists develop and incorporate new features to improve the simulation model. Parallelization of a modified code may not

follow easily from the parallelization of the original code. For example, the scientists at the Pacific Northwest National Laboratory (PNNL) [21] have introduced a new parameterization scheme into MM5 that has posed significantly new challenges for parallelization.

KCS has developed a new approach to automatic parallelization and built a prototype tool called the ParAgent [2][11]. The new approach is targeted at specific types of programs characterized by the underlying numerical method. The current version of ParAgent is applicable to time-marching explicit *finite difference method* (FDM) programs. MM5 is an example of such a program.

ParAgent is applied for one of the best known and widely used mesoscale climate models, the NCAR/Penn State MM5. The MM5 is formidable program to parallelize. Its manual parallelization took a 4-person team more than three years. A new version of MM5 recently developed at the PNNL is also parallelized using ParAgent. The modifications in the PNNL version are significant to the extent that a different parallelization strategy had to be adopted. A 2-D parallelization is done for the original MM5 whereas a 1-D parallelization is done for the PNNL version. The ParAgent generated parallel code for PNNL version of MM5 automatically. Along with parallel code generated by ParAgent, communication library was written to take care of data

decomposition, neighborhood communication, etc. While checking the output, several bugs were found and correcting them took slightly over two months by one person.

2. RELATED WORKS

In its complete generality, automatic parallelization is known to be NP-complete [3]. Research on parallelizing compilers has shown that it is difficult to achieve efficient parallelization using completely automatic code generation tools. The applicability of such tools to real-life problems has not been realized. The research focus has now shifted to interactive tools to help users during the parallelization process. ParAgent is a domain-specific semi-automatic parallelization tool, which not only depends on the user interaction but also incorporates an expert system type approach that employs domain-specific knowledge during parallelization. The current version of ParAgent specifically targets the Finite Difference Method (FDM). FDM is widely used in scientific computing. For example, several climate simulation models are based on FDM. ParAgent works in three phases. First, during the diagnosis phase, it checks the array access patterns in a given sequential code to ensure its consistency with the FDM model. This is an interactive phase where the user can provide additional information or corrections to resolve inconsistency problems or ambiguities detected by ParAgent. Second, during the communication analysis phase, it locates communication points and optimizes communication by merging and grouping the communication. Third, during the code generation phase, it performs index transformations and inserts communication to

generate a Single Program Multiple Data (SPMD) parallel program.

Although the conceptual structure of the FDM model looks simple, in practice, it is a formidable task to parallelize a legacy code. For example, it took a team of four people more than three years to parallelize NCAR/Penn State MM5. NCAR/Penn State MM5 consists of 164 subroutines and about 30,000 lines of code. The code is quite complex. As an extreme example, the main subroutine *solve1* includes a loop that spans more than a thousand lines of code with many levels of nested subroutine calls. The parallel codes are tested on a 64-node Pentium Pro 200 MHz PC clusters using a Fast Ethernet.

Table 1. Execution Time (sec) of 2-D Parallel MM5 on 64-node PC Cluster

Number of Processors	Grid Size					
	32 x 32 x 23			64 x 64 x 23		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	2760			18000		
4	750	3.68	0.92	4520	3.98	0.99
16	225	12.3	0.77	1350	13.3	0.83
64	120	23	0.36	420	42.8	0.67

The execution timings are reported in Table 1 for the 2-D Parallel MM5. The timings are given for two problem sizes. To obtain optimal benefit from parallel computing the problem size must be sufficiently large. In general, the speedup increases as the problem size is increased. On 64 processors, the speedup increases from 23 to 42.8 as the problem size is increased. The efficiency (speedup divided by the number of processors) is

higher for a smaller number of processors. For example for the smaller problem, the efficiency on 64 processors is 0.36 where as on 4 processors the efficiency is 0.92.

Library-based tools, such as the Runtime System Library (RSL) [12] and the Nearest Neighbor Tool (NNT) [4], are built on top of the lower level libraries, such as MPI, and serve to relieve the programmer of handling many of the details of message passing programming. Performance optimizations can be added to these libraries that target specific machine architectures. While a number of models were successfully parallelized using NNT [26], the serial and parallel versions of the code were distinctly different and had to be maintained separately. Further, parallelization is time-consuming and invasive, since code must be inserted by hand and the user is still required to do dependence analysis themselves. Source translation tools have been developed to help modify these codes automatically. One such tool, the Fortran Loop and Index Converter (FLIC), generates calls to the RSL library using command line arguments to identify decomposed arrays and loops needing transformations [5]. While useful, this tool has limited capabilities. For example, it was not designed to handle multiple data decompositions, inter-process communications, or nested models.

3. SOFTWARE ENVIRONMENT

KCS provides a comprehensive software environment to facilitate parallel computing on PC clusters. Currently, it includes ParAgent [11], WatchMon [25], Vis-5D [30], and Rabbit [31].

ParAgent is a domain-specific, interactive tool for automatic parallelization. The current version is applicable to time-marching, explicit FDM programs. The input is a Fortran 77 serial program and the output is a parallel Fortran 77 program with primitives for *message passing interface* (MPI) [22]. The output code is portable to distributed memory platforms. ParAgent follows a knowledge-based approach that uses specific characteristics of the FDM to guide the automatic parallelization process. ParAgent is used to diagnose, analyze and parallelize serial programs. It is an interactive tool with a graphical user interface to help the user. It shows the *call order tree*, the block-level *abstract syntax tree* (AST) and it provides a *variable tracking* facility to categorize and analyze a multitude of variables. After parallelization, the GUI can be used to see the sync/exchange points and the communication patterns using stencil-exchange diagrams and provides a block-level view to show how the parallel program relates to the serial program.

In addition to the tools being developed at KCS, the software environment includes the following tools: WatchMon [25], Vis-5D [30], and Rabbit [31]. The WatchMon is a web-based program to monitor the utilization of resources at each node in the cluster. The performance is shown in the form of a matrix. Each row of the matrix corresponds to one node in the cluster. The columns of the matrix show various performance related statistics including the utilization of the processor, the memory, and the swap space. This makes it easy to check the load balance on each node. The Vis-5D is a visualization tool. It can also be used to compare the outputs from the serial and the parallel program runs. The Rabbit is a tool for detailed performance analysis. It can be used to measure the instruction count, the number of floating point operations, the number page faults, the total number of memory accesses, and the number of first (or second) level cache accesses. The tool is based on the performance counters implemented in Pentium Processors.

The tools described here are used along with the LINUX operating system, the MPI library, and the PGF compiler [32] that is available from the Portland Group. The PGF compiler is useful to compile legacy codes that often have non-standard FORTRAN extensions such as the Cray pointers.

4. PARALLELIZATION OF MM5 PROGRAMS

MM5 code consists of about 57000 lines and 220 subroutines. Its conceptual structure is simple. It has an outer time-marching loop. The 3-dimensional cubical spatial domain is divided into grid cells. Inside the time loop, there can be several hundred nested loops spanning the spatial domain. These loops implement the physics and the dynamics associated with the physical phenomenon. The dynamics leads to communication described by stencil-exchange diagram [12]. A computation at a grid cell can require data values associated with other cells in the first (immediate), or second nearest neighborhood.

Although, the conceptual structure of MM5 is simple, it is a formidable program to parallelize. The code is large with many subroutines, hundreds of variables, and complex control structure of the programs. The structure of the program is very complex and hard to trace. For example, it includes a loop that spans more than a thousand lines and with many levels of nested subroutine calls.

4.1 USE OF PARAGENT

Using ParAgent, the parallelization of a FDM program involves three major steps: diagnostic, communication analysis and optimization, and finally the automatic code

generation. During the diagnostic step the serial program is analyzed using ParAgent. The diagnostic phase is interactive. ParAgent identifies specific problems, and the user has to fix or keep track of those using the auxiliary information provided by the ParAgent. For example, the variable UJ1(I) is identified as a problem and its usage pattern is shown. In MM5, the UJ1(I) happens to be a cryptic name for U(I , J) with J=1. The user has to replace UJ1(I) with U(I , 1). In the next step, ParAgent performs data flow analysis to identify the sync/exchange points, the variables to be communicated and their communication patterns. The last step is the automatic generation of parallel program. It involves changes in array declaration to incorporate decomposition, global-to-local index transformation and insertion of communication primitives.

ParAgent can perform either 1-D or 2-D parallelization. A 2-D parallelization, along *I* and *J* dimensions, is done for the NCAR version of MM5. A 1-D parallelization, along *J* direction, is done for the PNNL version of MM5. The reason is for the PNNL version parallelization along *I* direction presents several problems. The PNNL sub-grid parameterization scheme uses the "height" function during physics calculations. Instead of two separate loops, one covering a spatial dimension (loop indexed by *I* in the NCAR version of MM5) and another covering the "height," the PNNL program uses one single loop index that covers both *I* and the "height." It leads to a complex pattern of indexing

arrays and makes it difficult to parallelize along *I* dimension. The complex pattern of indexing cannot be handled automatically by the ParAgent.

4.2 PARALLELIZATION STEPS

A. The first step is to determine the direction(s) for domain decomposition. The program mostly uses I, J, and K to span the domain along three dimensions. The ParAgent is used to evaluate the choices for 1-D or 2-D parallelization. The code segments leading to communication are identified by ParAgent. The model is currently being used to study the mountainous northwest part of United States. The wide variations in altitude of the land surface cause highly irregular distribution of elevation bands along the *J* direction. It causes load imbalance as the PNNL MM5 is parallelized along the *J* direction. Specifically, the values of *I* loop control parameters *mix* and *mht* have wide variations along the *J* direction. Thus, we opted for 1-D parallelization along the *J* direction.

B. The *n*-D (*n* varies from 2 to 4 in the given code) arrays are decomposed along the *J* dimension. For example, a 3-D array A[MIX,MJX,MKX] becomes (in the parallel code) A[MIX, JLOC, MKX], where JLOC = MJX/NPROC+GA if MJX is divisible

by NPROC(number of processors) and JLOC=MJX/NPROC+GA+1, where GA is ghost area, otherwise. Each decomposed array is padded with a *ghost area* while computing the local array size (JLOC). The *ghost area* serves as the storage location for the data received from a neighboring processor. The ghost area concept is often used in parallel FDM codes. There are hundreds of arrays in the code. ParAgent displays information (dimension, size, and the indexing pattern) about all the arrays and makes it convenient to select arrays for decomposition.

C. ParAgent does optimization to minimize the overhead of *stencil-exchange type* communication. The stencil-exchange type of communication results from the finite differencing scheme and it occurs between a processor and its neighbors. The minimization strategy assumes existence of ghost area to cache the messages for reuse. The detection and optimization of communication is the most difficult aspect of parallelization of a complex and large code. The optimization is critical for performance.

D. Loops (in this case J loops) are changed and the global-to-local index transformations are done by ParAgent to generate the parallel code. The communication is inserted as high-level primitives. These primitives are similar in format to the RSL calls for

stencil-exchange communication. A typical serial loop in an FDM code and the corresponding parallel code with library calls are shown as follows:

```
Do I=2,n-1
Do J=2,n-1
    A(I,J) = B(I+1,j)+B(I-1,J)+B(I,J-1)+B(I,J+1)+B(I+1,J-1)
           +B(I-1,J-1)+B(I+1,J+1)+B(I-1,J+1)
Enddo
Enddo
```

```
Call Decompose (A, B)
Call Neighbor_communication(B)
Do I=ifirst, ilast
Do J=jfirst, jlast
    A(I,J) = B(I+1,j)+B(I-1,J)+B(I,J-1)+B(I,J+1)+B(I+1,J-1)
           +B(I-1,J-1)+B(I+1,J+1)+B(I-1,J+1)
Enddo
Enddo
Call Append_Array(A, B)
```

E. The code generated by ParAgent is linked with an MPI-based communication library. The library routine handles the details such as packing slices of *n*-D arrays to be communicated into a contiguous buffer, sending the exchange buffer among the processors and unpacking the buffers back into the ghost area of the *n*-D array at the destination processor.

F. Finally, the output is verified to check the correctness. We found several bugs while doing this step and this step took most during parallelization. Sometimes, the error came out after 6 hour simulation, which made it quite difficult to debug the code because the simulation took over 30 minutes and we have to run it again after applying changes on the code. Some of Bugs found are shown in next section.

4.3 DEBUG

Out of parallelization steps, debugging took most of time consumed because we didn't have any tool to work with. The followings are bugs found during parallelization of PNNL MM5.

A. Arrays within function call

Ex) scr(i,2) = cvmgt(1.,msf(i,j-1),ind.eq.0)

ParAgent gives error on this statement during preprocessing the code and user has to comment this line before processing the file. After generating parallel code, this line uncommented by human. However, there's a communication occurring in *J* direction. Thus, this communication has optimized not by ParAgent, but by user.

B. Fake *J* loop

```
Subroutine a
Real abc(mkx)
Do j=1,mkx
    abc(j) = 0.
Endddo
End
```

```
Subroutine a
Real abc(mkx)
Do j=jfirst, jlast
    abc(j) = 0.
Enddo
End
```

The dimension of array "abc" is "mkx". However, ParAgent does loop index transformation because it only recognizes the loop as *J* loop. This causes the loop go 1 to JLOC instead of 1 to mkx.

C. Insertion of communication calls where there's no declaration of the array involved in communications

Ex)

```
Subroutine a
Call b
Call c
End

Subroutine b
Common /test/ abc(mjx)
Do j=1,mjx
    abc(j) = abc(j) + 1
Enddo
End

Subroutine c
```

```
Common /test/ abc(mjx)

Do j=1,mjx-1

    abc(j) = abc(j-1)

Enddo

End
```

```
Subroutine a

Call b


Neighbor_comm(abc)


Call C

End
```

Array "abc" is defined in subroutine "b" and has neighborhood communication inside subroutine "c". Thus, ParAgent optimizes this communication, pulls it out and places communication call between "call b" and "call c", which is correct. However, array "abc" is not declared within subroutine "a" because subroutine "a" doesn't use array "abc" at all. It causes compile time error. User has to either declare array "abc" within subroutine "a" or place the communication call inside subroutine "c".

D. Global summation

Ex)

```
Do j=1,mjx
        total = total + abc(j)
Enddo
```

This adds all elements of array "abc" to the constant "total". Supposed that the value of array "abc" increments by one starting from 1, then "total" becomes 1+2+...+mjx.

```
Do j=jfirst_,jlast_
        total = total + abc(j)
Enddo
```

(sol) MPI_Allreduce (total, MPI_SUM)

But this adds only local portion of array "abc" to "total" causing that all processors have local sum of array "abc". If "total" is used after the loop, the result becomes incorrect. The solution is using "MPI_Allreduce" with MPI_SUM. All processors collects "total" from other processors and using MPI_SUM, it adds them. ParAgent gives a warning message during diagnostic step but currently it doesn't generate correct parallel code. Thus, user has to keep track of this.

E. Initializing some variable if only array index is equal to some value

Ex)

```
Subroutine a
First = true
Do j=1,mjx
If(first)
        temp = abc(j)
        First = false
Endif
Enddo
End
Subroutine a
First = true
Do j=jfirst_,jlast_
If(first)
        Temp = abc(j)
(SOL)   MPI_Bcast(temp)
        First = false
Endif
```

```
Enddo

End
```

This subroutine copies the first element of array "abc" to "temp" but in parallel code, every processor tries to copy the first element of local array "abc" to "temp". Thus, this "temp" is possibly different from processors but the value is supposed to be identical across processors. The solution is broadcasting "temp" from the processor who computes the first element of the array to the rest of processors. ParAgent didn't find this so the output was incorrect so it required a lot of debugging effort. The user had to debug the program by divide and conquer strategy to find where the output went wrong.

5. PERFORMANCE RESULTS

The performance results for the two versions of MM5 are reported in this section. The parallel programs were run on the following machines: a 64-node Pentium Pro 200 MHz cluster with fast Ethernet switch [23], a 16-node Pentium II 450 MHz cluster with Dolphin network [28], and an 8-node Pentium II 500 MHz cluster with the Giganet network [29].

5.1 PNNL MM5

The Table 2 shows the execution timings for the 1-D parallel algorithm for the PNNL version of MM5. We see that the performance is quite good for a small number of processors and it becomes worse as the number of processors increases. The problem is that the grid size of 37x37x23 is not large enough to run the problem on more than eight processors. The cluster with the Dolphin network performs significantly better.

Table 2. Execution time, speedup, and efficiency for the PNNL MM5

Proc	8-node 200 MHz Pentium Pro with Fast Ethernet			8-node 450 MHz Pentium II with Dolphin Network			8-node 500 MHz Pentium II with Giganet		
	Time	Sp	Eff	Time	Sp	Eff	Time	Sp	Eff
1	2738	1.00	1.00	1250	1.00	1.00	1156	1.00	1.00
2	1528	1.79	0.89	656	1.90	0.95	628	1.81	0.92
4	842	3.25	0.81	383	3.26	0.81	375	3.09	0.77
8	501	5.46	0.68	241	5.18	0.65	251	4.60	0.58

5.2 PERFORMANCE EXPERIMENTS WITH A FDM BENCHMARK

The parallelization of two versions of the well-known mesoscale climate clearly shows the feasibility of using PC clusters for an important application area. We believe that PC clusters will be even more effective when large grid sizes are used. If so, it will be possible to do more accurate analysis. In this section we present a study that shows how the performance changes with the problem size. We also compare the performance of PC clusters and other computing platforms. This study uses a FDM benchmark that is easier to experiment with it.

The FDM benchmark, given to us by a colleague from Dayton University, is program for an electromagnetism application. It is written in FORTRAN with 1443 lines. Unlike MM5, it is easy to change the problem size, requiring only a modification of a parameter. It is also easy to compile the benchmark code on different computing

platforms, as the code does not include any non-standard extensions of FORTRAN. Like MM5, the benchmark is a three dimensional, time marching, explicit finite difference method (FDM) program. It has communication characteristics similar to the MM5. For the benchmark program, the memory required per node depends on the parameters IL, KL and JLOC. The parameters IL and KL are fixed. The parameter JL is varied to change the problem size. We have done a 1-D parallelization using ParAgent and JLOC = JL/N is the size of a distributed array along the direction of parallelization, and N is the number of nodes.

5.2.1 PROBLEM SIZE AND PERFORMANCE VARIATIONS

Parallel computing makes it possible to run problems so large that it is not feasible to run them on a single node. For such large problems, the speedup with respect to a single node cannot be actually measured. We have introduced the metrics *normalized speedup* (NSP) and the *incremental speedup* (ISP) to deal with this situation. The NSP is obtained by multiplying by M the speedup of N nodes with respect to M nodes. The NSP coincides with the usual definition of speedup if $M = 1$. The number M increases with N. In case of a large problem, we select the smallest possible value of M, smaller than N but large enough so that the problem fits into the aggregate memory on M

machines without causing excessive paging. The NSP factors out the effect of paging and also allows speedup comparisons using large problems. The *incremental speedup* (ISP) shows the speedup relative to half the number of nodes. ISP can be viewed as a practical indicator of scalability. In scalable system, the ISP value should remain fairly constant when the number of nodes is doubled along with the same increase in the problem size.

A large pool of aggregate memory in a cluster can significantly improve the performance of a parallel program and lead to super-linear speedups. We have used the Rabbit tool to observe the dynamic behavior of the program including the caching and paging. We observed that the super-liner speedup is due to excessive paging. This is analyzed and confirmed as follows. It is observed that the excessive paging begins when the memory required per node exceeds a certain threshold value. For example, on one node the number of page faults varies in the narrow range between 173 and 175 when JL is less or equal to 122, but this number increases to 2352920 when JL = 244. Memory required per node in bytes is $(89*IL*JLOC*KL+76*JLOC*KL)*8$.

Table 3. Execution time(sec), Speedup, Normalized and incremental speedup

Proc	JL = 61			JL = 122		
	Time	SP	ISP	Time	SP	ISP
64				42.40	14.67	0.99
32	30.72	9.63	1.20	42.27	14.72	1.53
16	36.76	8.05	1.71	64.95	9.58	1.68
8	62.76	4.71	1.56	108.99	5.71	1.68
4	97.68	3.03	1.66	182.69	3.40	1.80
2	161.97	1.83	1.83	328.38	1.89	1.89
1	295.9			622.04		
Proc	JL = 244			JL = 488		
	Time	SP	ISP	Time	SP	ISP
64	60.57	23.35	1.29	84.89	38.82	1.61
32	77.91	18.15	1.54	136.55	24.13	1.67
16	120.12	11.77	1.67	228.67	14.41	1.86
8	200.72	7.05	1.84	424.61	7.76	1.94
4	369.12	3.83	1.92	823.88		
2	707.05	AH	AH			
1	31370					
Proc	JL = 976			JL = 1952		
	Time	NSP	ISP	Time	NSP	ISP
64	164.83	49.85	1.70	393.44	57.77	1.82
32	280.85	29.26	1.87	717.12	31.70	1.98
16	524.06	15.68	1.96	1420.7		
8	1027.1					

The results on the benchmark FDM program are reported in Table 3 using the three metrics for speedup. The notation "AH" indicates the values for the speedup are abnormally high. The instance of abnormally high speedup occurs when $JL = 244$ and the number of nodes is 2. The largest problem ($JL = 1952$) requires about 9.2 GB of

memory. To deal with the abnormalities, the NSP metric is used instead of the ordinary speedup (SP) starting with the case JL =244. Clearly, a large number of nodes can be used very effectively to solve large problems. This can be seen quantitatively using the speedup metrics.

On 64 nodes, the speedup increases from 14.67 to 57.77 as the problem size is increased. Also, the ISP values are close to the ideal value of 2 as the number of nodes and the problem size are both doubled. The ISP value is 1.83 for $NP = 2$ and $JL = 61$. After the same proportional increment in the number of nodes and the problem size it remains fairly constant, and finally it is 1.82 for $NP = 64$ and $JL = 1952$. These results show that the performance scales very well if the problem size is sufficiently large to take advantage of a large number of processing

5.2.2 COMPARISON OF PC CLUSTERS AND OTHER PLATFORMS

Another important point we would like to make is that clusters provide quite good performance compared to commercially available parallel computing platforms. Using the FDM benchmark, we compared the performance of SGI Origin 2000, IBM SP-2, a Pentium Pro 200 MHz cluster, and a Pentium II 450 MHz cluster. The number of nodes is eight in each case.

The results are shown in Figure 1. For each platform, the histogram shows the execution time for different sizes of the problem. It is seen that the 450 MHz Pentium II cluster performs the best except in the case of the smallest problem size. The SGI machine performance the best for the smallest size problem, however, it lags behind the Pentium II cluster as the problem size increases.

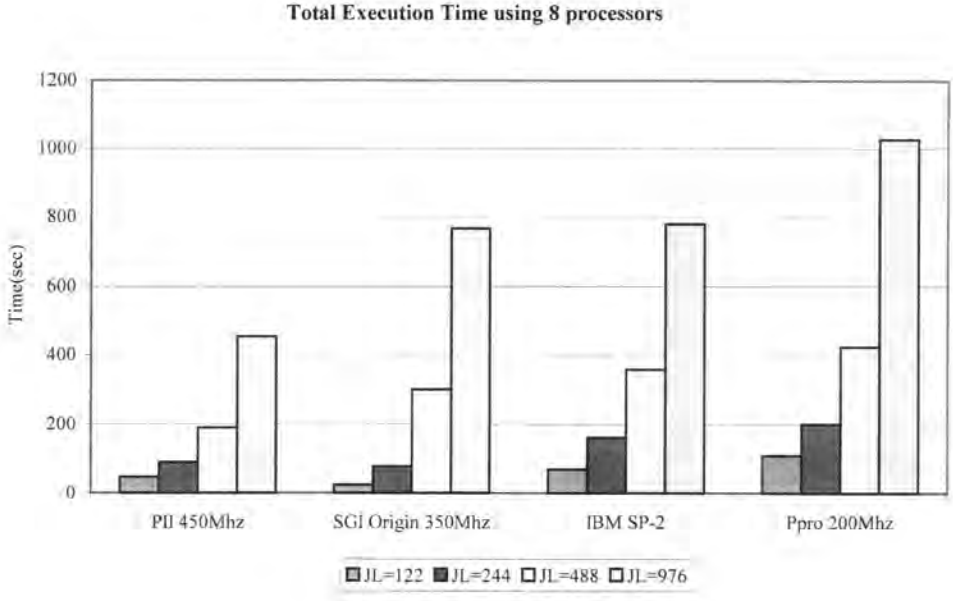

Figure 1. Comparison of PC Clusters and Other Computing Platforms

6. LOAD BALANCE

The load imbalance is the main problem in the case of the PNNL MM5. The PNNL scientists have developed a new sub-grid parameterization scheme that improves the model and leads to a more accurate prediction of snow. The model is currently being used to study the mountainous northwest part of United States. The wide variations in altitude of the land surface cause highly irregular distribution of elevation bands along the *J* direction as shown in Figure 2.

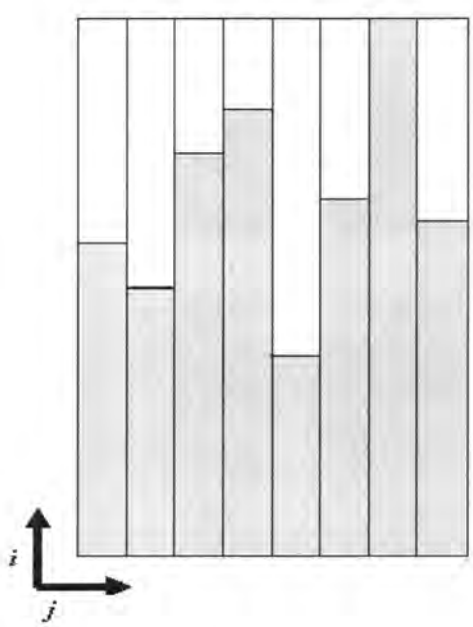

Figure 2. Structure of an unbalanced array

It causes load imbalance as the PNNL MM5 is parallelized along the *J* direction. Specifically, the values of the loop control parameters *mix* and *mht* have wide variations along the *J* direction, which makes it practically impossible to distribute the load completely uniformly. The results presented here are based on the best possible distribution that we could achieve.

As mentioned in section 4.1, PNNL version of MM5 uses loop index *I* for both a spatial dimension (loop indexed by *I* in the NCAR version of MM5) and the "height". This complex pattern of indexing made us to do 1-D parallelization along *J* direction and caused load imbalance. As shown in Figure 2, if we divide the array evenly along *J* direction among processors, some processors have more workload than others because the height of *I* direction varies.

The execution time, on an individual processor, for the main computation loop of MM5 is modeled fairly accurately by a linear expression $a \times mix + b \times mht$. The results for the 4-node cluster with the Dolphin network are shown in Table 4. The load imbalance is worse when eight nodes are used. Similar load imbalance behavior is observed on the three clusters. The communication time is not a significant overhead. For example, on the 4-node cluster with the Dolphin network, the communication time is less than eight seconds.

Table 4. Analysis of load imbalance due to irregular distribution of elevation bands

Processor	*mht*	*mix*	Exec time for the main loop	Estimated exec time = 0.1**mht* + 0.41**mix*
1	621	518	260	260
2	539	148	216	222
3	614	259	254	254
4	564	407	242	235

In Figure 3, it shows speedup. The left bar is the speedup when we decompose arrays evenly along J direction regardless of irregular distribution of elevation bands. The right bar shows the speedup after we distribute arrays so that workload on every processor is balanced.

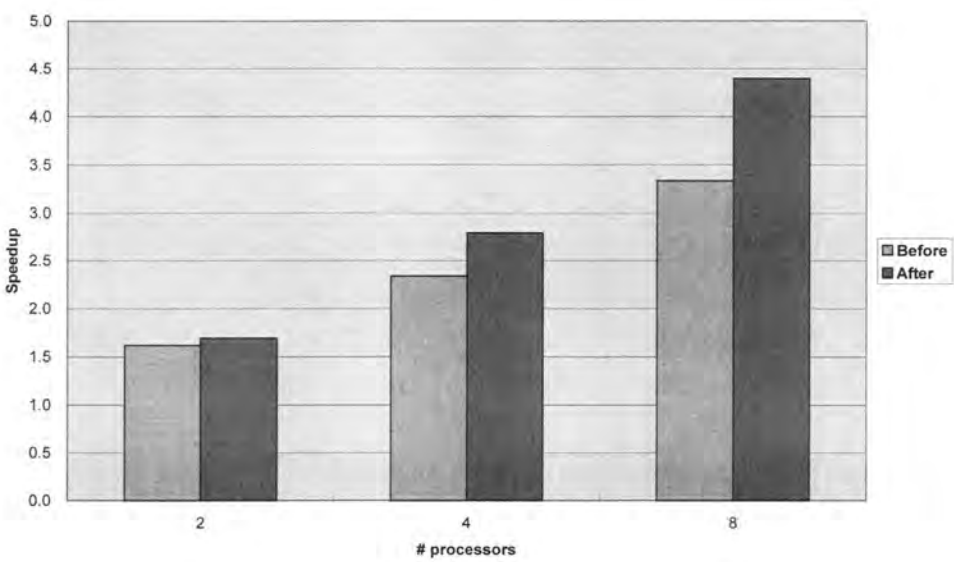

Figure 3. Speedup before and after reducing load imbalance

7. COMMUNICATION LIBRARY

A library is necessary for parallelizing serial code. Automatic parallelization tool, ParAgent, generates parallel FORTRAN code automatically. This includes loop index transformation, data decomposition, neighborhood communication, etc. These changes work with proper library. This library is for distributed memory systems and useful in parallelizing *finite difference method* (FDM) programs such as MM5.

Loop index transformation changes global loop index to local index. Data decomposition reduces the original array size so that each processor computes only small portion of the array. Neighborhood communication occurs when a processor depends on data from nearest neighborhood processor, which is one of the properties of FDM programs.

This library makes it easy to parallelize serial FDM code with least changes on parallel code generated by ParAgent. It mainly depends on Single Program Multiple Data (SPMD), Message Passing Interface (MPI) and Fortran 77, and handles both 1-D and 2-D parallelization.

7.1 DATA DECOMPOSITION

There are two types of data decomposition: cyclic and block decomposition. As shown in Figure 4, the former divides data to each row and assigns it to each processor alternatively. Thus, row 1 and 3 are assigned to processor 0 and row 2 and 4 to processor 1. The latter divides data to as many blocks as the number of processors. If there are two processors and 4 rows, processor 0 gets the first two rows and processor 1 gets the last two rows.

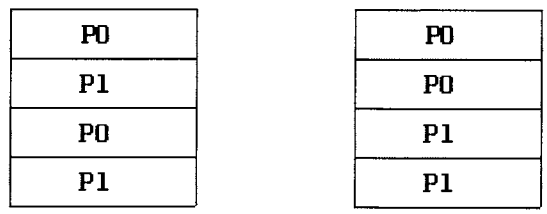

Figure 4. Cyclic and Block Decomposition

In our serial FDM codes, we have found lots of neighborhood communications that requires data from adjacent processors. In figure 4, we observe that cyclic decomposition requires two rows of data while block decomposition needs one row and it will require more data as the number of rows increases. Therefore, block decomposition is more appropriate for FDM and it is used in our library for data decomposition.

The data has to be decomposed properly to take an advantage of parallel computing. It will affect not only workload on each processor, but also performance depending on how we divide the data. For PNNL version of MM5, we use 1-D block decomposition along *J* index due to complexity of code as mentioned in section 4.1. Thus, the dimension of an array with *J* index will be changed from "A(JMAX)" to "A(JLOC)", where JMAX is the maximum value of *J* index and JLOC is the ceiling of JMAX/(the number of processors) plus the size of ghost area. We divided *J* index as evenly as we can among processors but found load imbalance because of mixed *I* loop. It deteriorated the performance of parallel program as described in section 6.

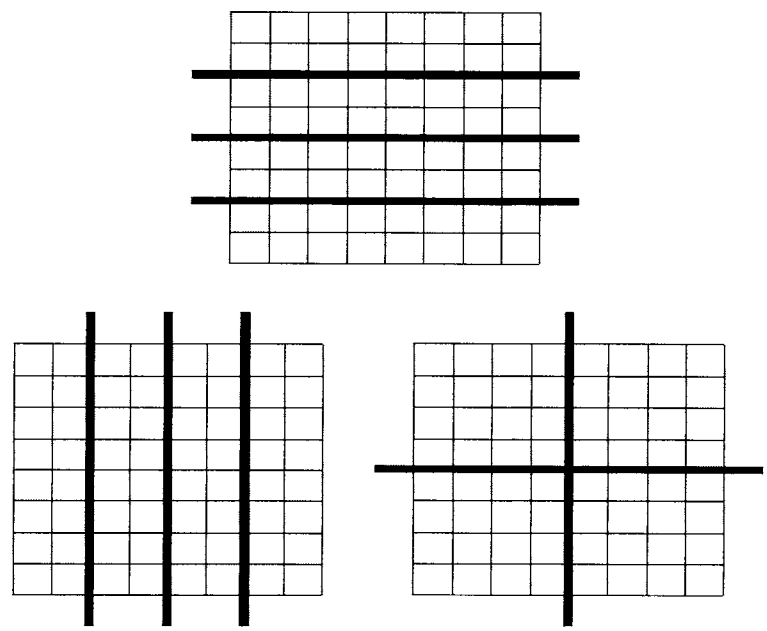

Figure 5. Examples of Block Decomposition

In Figure 5, three types of data decomposition are shown. They cause data dependency between neighboring processors when a processor computes boundary cells.

A. We can divide an array by the number of rows. For example, if the array size is 8x8 and the number of processors is 4 then each processor will be assigned two rows of the array and 8 columns (2x8). However, the local array size becomes (2+GA)x8 because we need ghost area for neighborhood communication.

B. Arrays can be decomposed along columns so that each processor is assigned 8 rows and 2 columns 8x(2+GA).

C. We can mix A with B. Then, each processor will have 2 rows and 2 columns (2+GA)x(2+GA). This is used for 2-D parallelization while A and B are for 1-D parallelization.

7.2 LOOP INDEX TRANSFORMATION

Once data is decomposed, the FORTRAN loops must be transformed so that each processor computes data local to itself. For example, if *J* loop goes 1 to 12 and there are 4 processors, then *J* loop after parallelization should go 1 to 3 on all 4 processors. We use "idx1first" and "idx1last" to handle this. Thus, local *J* loop goes "idx1first" to "idx1last" and their values are 1+GA and 3+GA, respectively, if we put the ghost area in the picture.

If it is 2-D parallelization, we also use "idx2first" and idx2last" for the second parallelizing index.

There are two types of loops. The first loop goes 1 to JMAX, where JMAX is the maximum value of the loop index and the other one goes 1+*c* to JMAX-*d*, where *c* and *d* are constants whose values are greater than 0 and less than JMAX. The following shows how these two loops are modified:

J=1,JMAX → J=idx1first,idx1last

J=2,JMAX-1 → J=idx1first(2),idx1last(JMAX-1)

"idx1first" and "idx1last" are constant variables but "idx1first()" and "idx2last()" are function calls defined in the library. Their input is the global index between 1 and JMAX, and output is the local index between idx1first and idx1last.

7.3 DISTRIBUTION OF ARRAY

The original array must be distributed among processors when the program reads an array from input file or uses a pointer to get values from other arrays. For example of "READ(60) A", array "A" reads data from file unit 60. This array is rather global than local because the file read requires an array with the size before decomposition so it has to be distributed among processors depending on decomposition scheme applied. We use

either a temporary array or a library call "DIST_ARRAY" because array "A" is already decomposed in the parallel code generated ParAgent or "PREAD", which is one of library calls. The changes are as follows:

READ(60) A → READ(60) TEMP_A

CALL DIST_ARRAY(TEMP_A, A)

READ(60) A → CALL PREAD(60)

The routine, "DIST_ARRAY", divides the source array into small arrays depending on the workload of each processor and distributes them to each processor so that each processor computes only some portion of the original array.

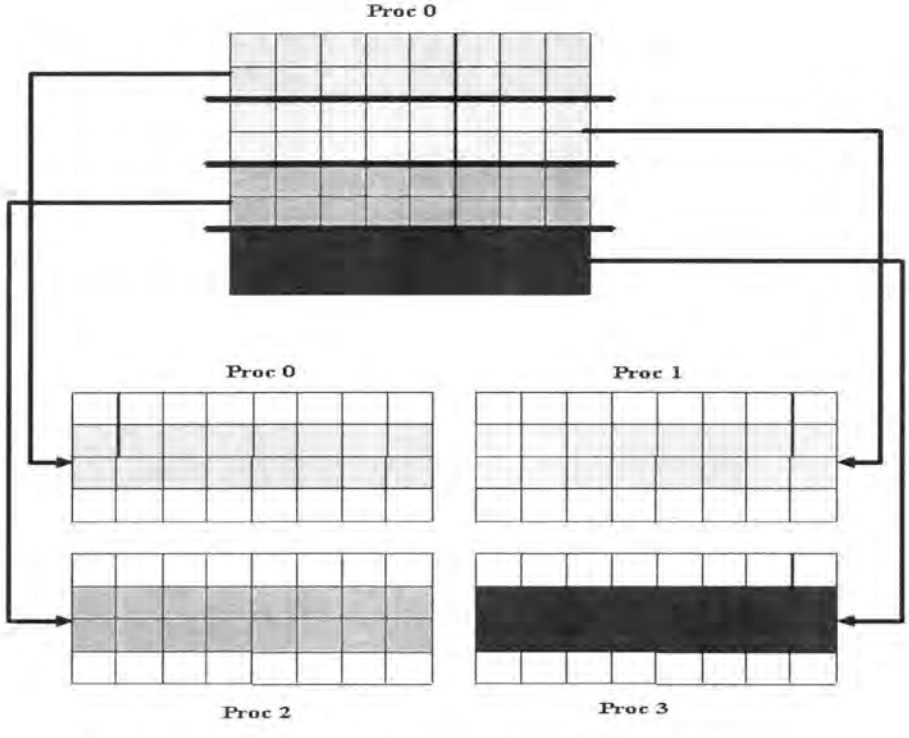

Figure 6. Distributing an array for 1-D Parallelization

In Figure 6, it shows how an array is distributed to processors. In case of an 8x8 array for 1-D parallelization, P0 divides the array into 4 2x8 arrays and distributes them to all processors. Also, each processor has ghost areas for neighborhood communication. If communication distance is 1, we add 1 slice for each side, which makes the size of ghost area 2. If it is 2, then ghost area is 4. Thus, in our example, the size of local array becomes 4x8 instead of 2x8 but it computes only two rows.

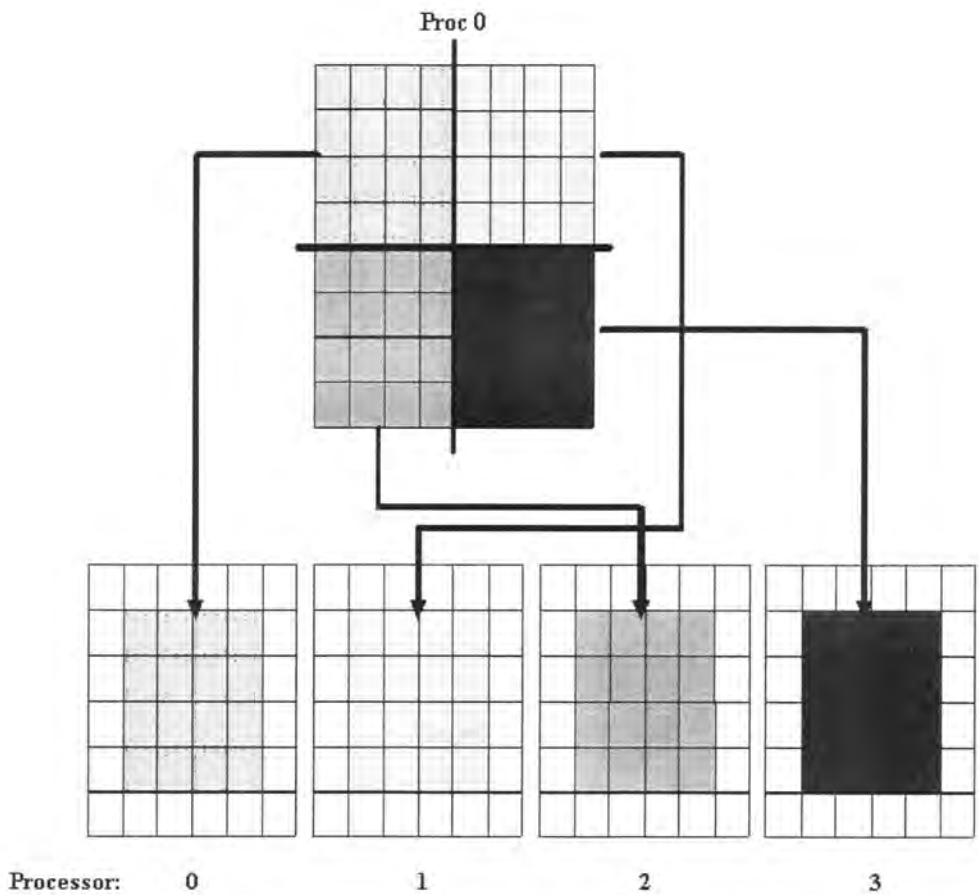

Figure 7. Distributing an array for 2-D Parallelization

In case of 2-D parallelization, P0 divides the array into four 4x4 arrays and distributes them to all processors in Figure 7. Notice that on each processor the small array size is 6x6 instead of 4x4 because we need ghost area for nearest neighborhood communication.

7.4 NEIGHBORHOOD COMMUNICATION

Data dependencies occur between neighboring processors quite often in FDM code. To resolve them, some data need to be copied to another processor. Each processor decides which slice to send up to three processors since there are three neighboring processors: horizontal, vertical and diagonal neighbor. Also it receives data from maximum three processors and copies it into ghost area.

Currently, maximum communication distance handled by this library is 2. There are 2 communication directions, either E, W or S, N for 1-D parallelization and 8 directions, E, W, S, N, SE, SW, NE, NW for 2-D parallelization.

In the Figure 8, each processor sends the last row of its local array to the processor in South and receiving processors copy data into ghost area above the first row of actual local array. The data in ghost area is later used when communication occurs in the loop to get rid of data dependency between processors.

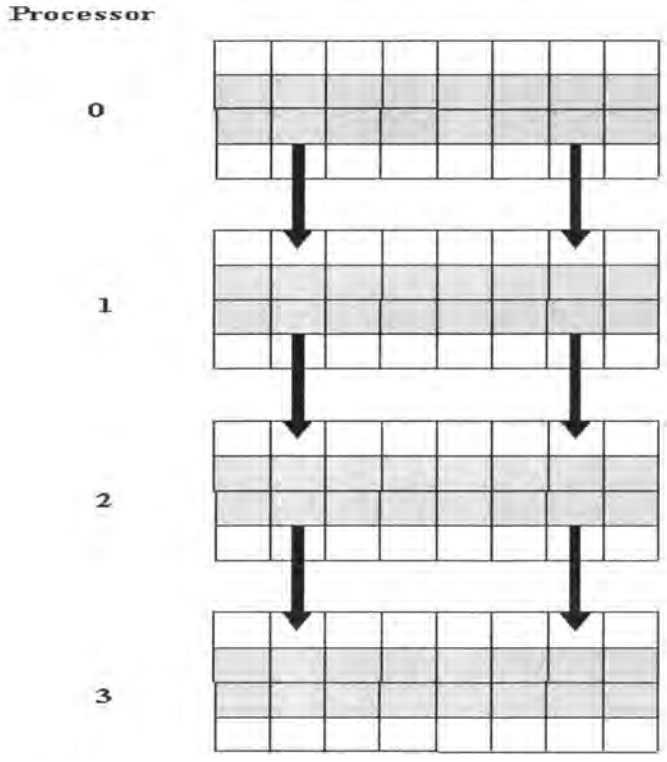

Figure 8. Neighborhood communication for 1-D Parallelization

Figure 9 shows that communication of 2-D parallelization is more complex than that of 1-D communication because some processors send/receive data to/from multiple processors while a processor sends/receives data to/from only one processor in 1-D parallelization. For example, P1 sends data to three different processors, P0, P2 and P4. Likewise, P2 receives data from three processors, P0, P1, and P4.

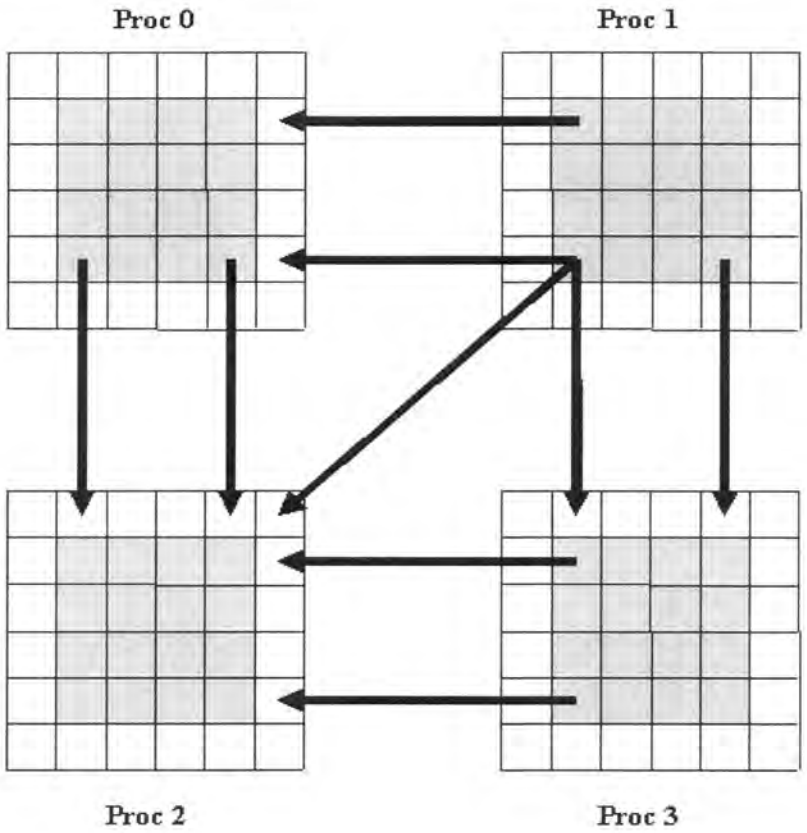

Figure 9. Neighborhood communication for 2-D Parallelization

Example:

```
Do I=2,n-1
Do J=2,n-1
        A(I,J) = B(I+1,J+1)+B(I-1,J-1)
Enddo
Enddo
```

⇩

```
Call Neighbor_comm(B, idx1loc, idx2loc, kdim, ldim, 1, 2, 1, 1)
Call Neighbor_comm(B, idx1loc, idx2loc, kdim, ldim, 1, 2, 1, 1)

Do I=idx1first(2), idx1last(n-1)
Do J=idx2first(2), idx2last(n-1)
```

```
            A(I,J) = B(I+1,J+1) + B(I-1,J-1)
        Enddo
        Enddo
```

In this example, array "B" has data dependency with processor in north-east and south-west. In parallel code, "Neighbor_comm" routine gets data from those processors and copies it to ghost area. Also, the loop index has been changed by calling functions which changes global index to local index because the loop goes 2 to n-1. Finally, array "A" computes the statement using data in the ghost area of array "B".

7.5 APPENDING ARRAY

While "Dist_array" is used for mostly parallel file read, "Append_array" is used for parallel file write. This appends the local arrays on each processor into an array whose size is the same as the size of the original array before parallelization. This is needed for mostly writing outputs since output requires a whole array instead a small local array.

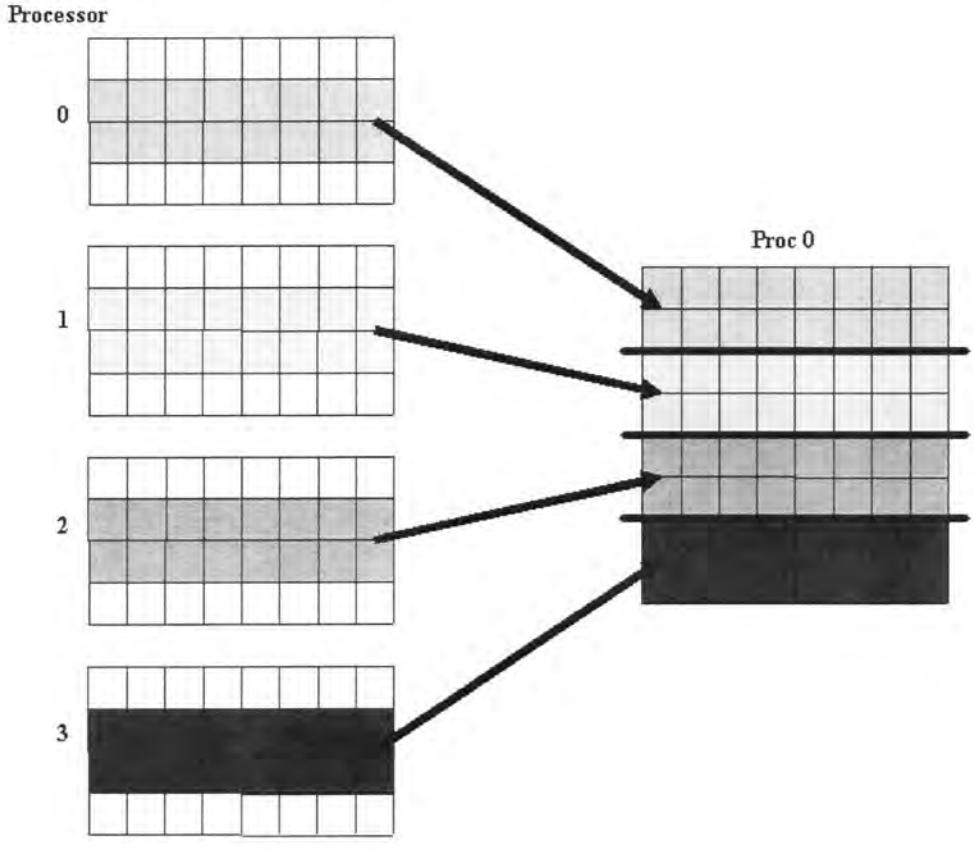

Figure 10. Appending an array for 1-D Parallelization

In the Figure 10, each processor sends its local array (2x8) to processor 0 and processor 0 appends them. The result can be used for output in the program or checking correctness of parallel program. This routine uses "MPI_Gathers" to collect local arrays on each processor.

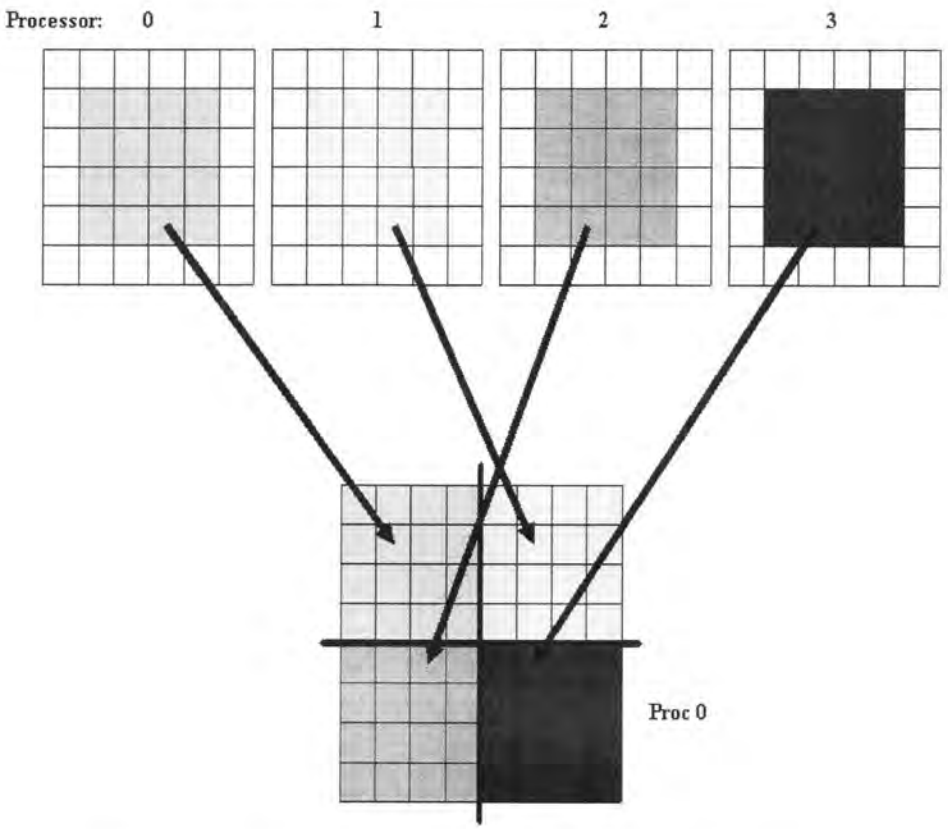

Figure 11. Appending an array for 2-D Parallelization

In Figure 11, each processor sends a 4x4 local array to processor 0, and then processor 0 appends them into original position before decomposition.

8. CONCLUSIONS

Cluster computing provides an inexpensive but high-performance alternative for advanced climate modeling research. PC cluster and our software environment makes easy to parallelize other legacy codes for modeling land surface, ocean, ground water, acid rain, and air pollutants. While cluster computing is attractive, its use will be limited without appropriate software tools. Along with ParAgent, we have used a number of other tools that facilitate cluster computing for real-life computation problems. They are good additions to solving complex problems quickly and more accurately.

REFERENCES

[1] Kothari, Cho, Deng, et. al. Software Tools and Parallel Computing for Numerical Weather Prediction Models, Third International Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications, Fort Lauderdale, Florida. April 2002.

[2] Deng, et. al. ParAgent – A Software Reengineering Tool for Parallel Computing, Twelfth IASTED International Conference on Parallel and Distributed Computing System, Las Vegas, November 2000.

[3] J. Li and M. Chen. Data Alignment Phase in Compiling Programs for Distributed Memory Machines. Parallel and Distributed Computing, 13:213-221, 1991.

[4] B.Rodriguez, L.Hart and T.Henderson, Parallelizing Operational Weather Forecast Models for Portable and Fast Execution, Journal of Parallel and Distributed Computing, 37, 159-170, 1996.

[5] J.Michalakes, FLIC: A Translator for Same-Source Parallel Implementation of Regular Grid Applications, Tech. Rep. ANL/MCS-TM-223, Argonne National Laboratory, 1997.

[6] R. A. Anthes, and T.T. Warne. Development of Hydrodynamic Models Suitable for Air Pollution and Other Mesometeorological Studies. *Mon. Weather Review*, #106, pp. 1045-1078, 1978.

[7] Cheng, *A Survey of Parallel Programming Languages and Tools*, Tech. Rep. RND-93-005, NASA Ames research center, Moffet Field CA 94035, 1993.

[8] Peter Corbett, et. al. Overview of the MPI-IO parallel I/O interface. *Input/Output in Parallel and Distributed Computer Systems*. Kluwer Academic Publishers. 1996.

[9] J. Dongarra, and B.Tourancheau, Editors. Environments and Tools For Parallel Scientific Computing, Advances in Parallel Computing, Vol. 6, Elsevier Science Publishers, 1993.

[10] F. Knop and V. Rego. Parallel Labeling of Three-Dimensional Clusters of Networks of Workstations. *Journal of Parallel and Distributed Computing,* 49:182-203, 1998.

[11] S. Kothari and S. Mitra. Parallelization Agent: A New Approach to Parallelization of Legacy Codes. *Eight SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[12] J. Michalakes. 1997c: RSL: A Parallel Runtime System Library for Regional Atmospheric Models with Nesting. preprint *ANL/MCS-P663-0597*.

[13] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File Access Characteristics of Parallel Scientific Workloads. *IEEE Transactions on Parallel and Distributed, Systems*, Vol 7, No. 10, 1996.

[14] P.S. Pacheco. Parallel Programming with MPI. Morgan Kaufmann Publishers, 1997.

[15] D. K. Panda and L. M. Ni. Special Issue on Workstation Clusters and Network-Based Computing. *Journal of Parallel and Distributed Computing*, 43:63-64, 1997.

[16] G. F. Pfister. In Search of Clusters. Prentice Hall, Englewood Cliffs, NJ. 1995.

[17] P. Steenkiste. Network-Based Multicomputers: A Practical Supercomputer Architecture. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 8, pp. 861-875, Aug. 1996.

[18] R. Thakur, E. Lusk, and W. Gropp. I/O in Parallel Application: The Weakest Link. *The International Journal of High Performance Computing Applications*, Vol. 12, No. 4, pp. 389-395, Winter 1998.

[19] R. Thakur, W. Gropp and E. Lusk. On Implementing MPI-IO Portably and with High Performance. *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pp. 23-32, 1999.

[20] J. Todd. Survey of Numerical Analysis. McGraw-Hill Publishers, 1962.

[21] Pacific Northwest National Laboratory: http://www.pnl.gov (May 2000)

[22] MPI Forum http://www.mpi-forum.org/docs/mpi-20-html (Aug 1999)

[23] Alice Cluster: http://www.scl.ameslab.gov (Aug 1999)

[24] SRL homepage: http://darwin.ee.iastate.edu/~srl/ (Aug 1999)

[25] WatchMon: http://darwin.ee.iastate.edu/cgi-bin/srl-monitor (Aug 1999)

[26] Communication library: http://www.fsl.noaa.gov (Jan 2001)

[27] Myrinet: http://www.myri.com/myrinet (May 2000)

[28] Dolphin: http://www.dolphinics.com (May 2000)

[29] Giganet: http://www.giganet.com (May 2000)

[30] Vis-5D: http://www.ssec.wisc.edu/~billh/vis5d.html (Aug 1999)

[31] Rabbit: http://www.scl.ameslab.gov/Projects/Rabbit/ (Aug 1999)

[32] Portland group compiler: http://www.pgroup.com (May 2000)